An Evaluation of the High Level Architecture (HLA) as a Technology for Use in Space Mission Simulations

Results and Conclusions from the SIMSS HLA Study and Prototyping Effort

Michael R. Reid

Computer Sciences Corporation

Consolidated Space Operations Contract (CSOC)

October 3, 2000

Executive Summary

The High Level Architecture (HLA) is a current U.S. Department of Defense (DoD) and an upcoming industry (IEEE-1516) standard architecture for modeling and simulations. It provides a framework and a set of functional rules and common interfaces for integrating separate and disparate simulators into a larger simulation. The main function of the HLA is to integrate and facilitate the interoperability of numerous different simulators running collaboratively on a variety of platforms and to provide advanced time and data management for the overall simulations. The HLA accomplishes this by defining a set of rules to which all compliant applications must adhere, a common model for sharing data, a standard schema for data documentation, and most importantly, an application program interface (API) for a Runtime Infrastructure (RTI) software system.

For data sharing among participant applications within a simulation, the HLA specifies an object-oriented variation on the basic *publish and subscribe* paradigm. Persistent data are stored within the attributes of HLA *objects*. Ephemeral data are distributed through HLA *interactions*, essentially broadcast messages. Data producing applications update HLA object attributes and send interactions. Data consuming applications read object attributes and receive interactions. Applications can be, and commonly are, both publishers and subscribers of data.

The HLA is especially well suited as a basis for game-like simulations in which a little “universe” is being simulated with multiple actors coming and going and interacting. It also has potential as a framework for integrating numerous separate simulators into a larger distributed simulation system. The fact that HLA-based simulators are designed around data classes from which any number of instances may be created give them great potential for easy expandability. The HLA’s inherent distributed nature should also allow for essentially unlimited scalability.

The HLA is not in itself, a simulator or a modeling tool. It is not a rapid application development system for simulations and it does not provide any data display capabilities or user interfaces. Although intended to make integrating disparate and remote simulators more practical, it will not make “plug and play” a reality. The HLA is probably not an appropriate architecture for simulations that just generate data for some single, external system or for simulations that cannot define their problem space as a collection of interacting objects. For systems such as these, the HLA would just be an unneeded extra layer and would provide little or no additional value. Simulators required to send data through a specific type of interface, such as IP or through a hardware bus are also not suited for the HLA, since bypassing the RTI for external communications violates one of the primary rules of the HLA. This is not to say that simulators such as these could not be formed into components of larger HLA-based simulations. They certainly could be.

The HLA could find a role in NASA as a core technology in certain types of space mission simulations, especially those involving multiple spacecraft flying in formation. It could also serve as the basis for the scientific modeling of natural systems. The HLA will not increase the fidelity of simulators, but its modular and distributed design will make it easier to build higher-fidelity simulators.

In order to evaluate the suitability of the HLA as a technology for NASA space mission simulations, a simulations group at GSFC conducted a study of the HLA and developed a simple prototype HLA-based simulator.

Abstract

The High Level Architecture (HLA) is a current Department of Defense (DoD) and an upcoming industry (IEEE-1516) standard architecture for modeling and simulations. It provides a framework and set of functional rules and common interfaces for integrating separate and disparate simulators into a larger simulation. The goal of the HLA is to reduce software costs by facilitating the reuse of simulation components and by providing a run-time infrastructure to manage the simulations. In order to evaluate the applicability of the HLA as a technology for NASA space mission simulations, the CSOC Simulations Group conducted a study of the HLA and developed a simple prototype HLA-compliant space mission simulator. This report contains a summary of the prototyping effort and a discussion of the HLA with the author's conclusions as to its strengths, limitations, and potential usefulness in the design and planning of future NASA space missions with a focus on risk mitigation and cost reduction.

Acknowledgements

The author wishes to acknowledge the important contributions made to this HLA study and prototyping project by his colleagues and fellow team members, Dr. Neil A. Ottenstein and William G. Parlock both of Computer Sciences Corporation.

Contents

Introduction

The High Level Architecture (HLA)[1] is a standard software architecture for modeling and simulations. The Defense Modeling and Simulation Office (DMSO)[2] of the U.S. Department of Defense (DoD) sponsored the development of the HLA as part of an effort to control the costs and increase the reliability of its simulators. Although originally designed to fit the extensive simulation needs of the DoD and other allied defense organizations, the HLA is by no means limited to military applications. It is already finding users well beyond the defense arena and will soon emerge as an IEEE standard (IEEE-1516)[3]. The HLA is a suitable core technology for diverse applications ranging from games designed purely for amusement to the serious simulation and modeling of complex and sophisticated industrial processes and work flows.

Modern simulation systems often reside on networks of computer systems with numerous individual simulators concurrently running on separate and disparate computing platforms. The concept of distributed and remote computing lies at the core of the HLA. Its main function is to integrate and facilitate the interoperability of numerous different simulators running collaboratively on a variety of platforms and to provide advanced time and data management for the overall simulations. The HLA accomplishes this by defining a set of rules to which all compliant applications must adhere, a common model for sharing data, a standard schema for data documentation, and most importantly, an application program interface (API) for a *Runtime Infrastructure* (RTI) software system. To put it succinctly, "The HLA is the glue that allows you to combine computer simulations into a larger simulation."[4]

In many ways, NASA's simulation needs mirror those of the DoD. Like the DoD, NASA operates complex operational systems and relies on simulators to reduce both risk and cost and will likely do so to an even greater extent in the future. Several recent mission failures, schedule delays, and cost overruns underscore the need for the high-fidelity simulation of missions prior to their launch. The HLA is best suited for game-like simulations, in which numerous "actors" come and go and interact. In the NASA realm, large simulation systems, which model the interplay of multiple spacecraft, ground systems, natural objects, and natural phenomena would be candidates for an HLA-based design.

During fiscal year 2000, the author, with assistance from colleagues, conducted a study of the HLA and developed a rudimentary, prototype space mission simulator based on this technology. The purpose of the effort was to evaluate the suitability of the HLA as a technology for NASA space mission simulations and scientific modeling. This paper introduces the HLA, describes the prototyping effort, documents the things that were learned, and presents the author's conclusions.

1 See http://hla.dmso.mil/.
2 See http://www.dmso.mil/.
3 See http://standards.ieee.org/.
4 Frederick Kuhl, Richard Weatherly, Judith Dahman, *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*, p.1, Prentice Hall PTR, 1999.

Overview of The High Level Architecture (HLA)

Synopsis

The HLA is a DoD and IEEE standard framework that supports modeling and simulations. It facilitates distributed and multi-platform computing by integrating disparate applications running concurrently at both local and remote locations. In fact, HLA-based simulations run best as distributed systems over a computer network. The RTI handles all communications between participating simulators, synchronizes the overall simulation, and provides data and time management. The HLA consists of a set of rules to which all compliant applications must adhere; an interface specification, which all compliant applications must exclusively use for external interactions; and a standard format for defining and describing shared data. The HLA is an architecture, not a simulator or modeling tool in itself.

For data sharing among participant applications within a simulation, the HLA specifies an object-oriented variation of the basic *publish and subscribe* paradigm. Persistent data are stored within the attributes of HLA *objects*. Applications that provide data publish the relevant attributes of the appropriate objects and update them. Applications that receive data subscribe to those attributes and read them. Ephemeral data are stored in HLA *interactions*. Data producing applications send interactions on a one-time basis and data consuming applications receive them. Applications can be, and commonly are, both publishers and subscribers of data.

Nomenclature

Like most technologies, the HLA has a specialized terminology associated with it. A list of the most important terms and their definitions follows. See the glossary at the end of this paper for a more expansive list.

Federate: An individual simulator application. These are the independent simulators, which the HLA integrates together into a larger collaborative simulation.

Federation: A simulation composed of two or more (often many more) federates integrated together.

Federation Execution: A session in which a federation is running, usually as a distributed system.

Federation Object Model (FOM): The common object model that defines the data shared between federates within the federation.

Simulation Object Model (SOM): The object model that defines the data shared by an individual federate with the federation. The FOM is a subset of the collection of SOMs defined for the federates in the federation.

Federation Execution Data (FED): The actual data defined in the FOM, which is shared between two or more federates within the federation execution.

Object: An HLA object is a container for data, which is created by a federate during the federation execution and persists for the duration of the federation execution or until a federate deliberately destroys it. Like objects in object-oriented programming languages such as C++ or Java, HLA objects are defined as classes and are realized as instances of those classes. They are composed of attributes, which can be of various data types, including composite data types. Subclasses of objects can be derived from other classes with full attribute inheritance and extensibility. HLA classes do not contain methods (a.k.a., member functions). Federates which wish to update some or all of the attributes of any given class of object, must *publish* those attributes. Federates which wish to read the data stored in an object's attributes, must *subscribe* to the attributes of interest. Only one federate at a time may own a given instance of an object; however, other federates may "own" some of the attributes of that object and publish them and different federates may own different attributes within that same object instance. Any federate may subscribe to the attributes of any class of object. All objects are defined in the FOM and any federate that wishes to publish or subscribe to an object's attributes must also define that object in its SOM.

Interaction: An HLA interaction is essentially a broadcast message that any federate within the federation execution can send or receive. Like HLA objects, interactions are data containers, which are defined as classes within the FOM and can be derived from other classes with full inheritance and extensibility. Federates publish them and subscribe to them. Unlike HLA objects, interactions do not persist. They are sent out and are either received or missed. If no subscribing federate receives the interaction, the data it carries are lost. For some reason, the data elements within interaction classes are called *parameters* instead of attributes. Any federate which wishes to publish or subscribe to an interaction must define that interaction in its SOM.

Runtime Infrastructure (RTI): The COTS or GOTS software, which implements the HLA interface and runs the federation execution (i.e., the overall simulation).

The Federation Object Model (FOM)

The Federation Object Model (FOM) describes the universe. That is, the universe that the federation is modeling. It is an abstract, but fundamental component of the HLA. The RTI and the other federates need a precise description of the type and form of every object and interaction that they share. The HLA defines a standard format and syntax for documenting objects and interactions, essentially a data description language. This data description language has two forms, a set of tables that are easy for humans to create and read based on the HLA Object Model Template (OMT) and a LISP-like syntax that is easy for the RTI to read.

The FOM defines every object and interaction known to the federation and provides a complete list of every attribute contained in every object and every parameter contained in every interaction. It also precisely specifies the forms and data types of all attributes and parameters. The data definition language provides the basic numeric and character data types, but also provides the means for one to define application-specific enumerated and complex data types (structures). The OMT tables are for humans to read and are the documentation of the FOM required for HLA-compliance certification. One writes the LISP-like syntax to the Federation Execution Data (FED) text file. The RTI reads the FED file and from it knows about the objects and interactions shared within the federation.

For HLA-compliance, every federate must have a Simulation Object Model (SOM) defined for it and documented in accordance with the OMT. The SOM is really just a miniature FOM that only deals with the data shared externally by that federate. The FOM is a subset of the aggregate of the SOMs of all of its member federates. It is a subset because some SOMs may contain data definitions not used in that particular federation. When designing a federation, the persons responsible for integrating the various federates convene to agree on the FOM in a meeting sometimes called the "FOM-o-rama."

DMSO provides a convenient tool for documenting FOMs and SOMs. The *Object Model Development Tool* (OMDT) makes it easy for a federation designer to create OMT-compliant tables. Especially useful in the case of FOMs, it will automatically generate an operationally usable FED file from those tables.

Services Provided by the RTI

The RTI provides six basic categories of services described below. A federate makes use of these services by making calls to the HLA RTI interface through its *Local RTI Component* (LRC). The LRC consists of an interface and a library to which the federate binds. As defined in the HLA specification, the LRC implements an *RTI ambassador*, which provides the application program interface (API) that a federate uses to send directives and information to the RTI. It defines a *federate ambassador*, which consists of a set of callback functions implemented for the specific federate. The RTI calls these functions to send directives and information to the federate. In the case of the DMSO RTI, the RTI ambassador and the federate ambassador allow the federate to communicate with the *fedex* and *rtiexec* daemons, which manage the federation execution and handle the network protocols.

Federation Management

The RTI incorporates the *Federation Execution Data* (FED) into the federation execution by reading the FED configuration file ("FED file"). It also creates federation executions, provides a means for an HLA-compliant application to join the federation execution as a participating federate and to resign from it, sets federation-wide synchronization points, effects saves and restores, and destroys the federation execution.

Declaration Management

The RTI provides this service to manage the publication and subscription to HLA objects and interactions. By calling the RTI's declaration management services through the HLA RTI interface, a federate can publish selected object attributes and interaction parameters and subscribe to them.

Ownership Management

Through this service, the RTI controls the ownership of object attributes. In order to update an object instance's attributes, a federate must first secure ownership of those attributes from the current owner. The current owner may either relinquish ownership to the requesting federate or refuse the request. The RTI handles this transaction. Only the owning federate may update attributes of the given object instance; however, not all of the attributes in an object instance need be owned by the same federate. Every HLA object inherits a special *privilegeToDeleteObject* attribute from the HLA-defined **ObjectRoot** base class. Only the owner of this attribute may delete the given object instance from the federation. By default, the federate, which creates and registers an object instance initially owns all of its attributes. The RTI enforces these rules.

Time Management

Through its time management service, the RTI controls when federates can advance their positions on the federation's time-line. This determines when they receive notifications of changes to the state of the federation execution, such as object attribute updates and when they receive time-stamped interactions. The RTI can manage either clock-driven (time-step) simulators or event-driven simulators. In the case of clock-driven simulators, each federate is responsible for maintaining its clock. However, in order to advance its clock, a federate must first request permission from the RTI and then wait for it to be granted. This allows the RTI to keep the federation synchronized. The RTI delivers time-stamped events such as object attribute updates and interactions to a federate only when that federate has reached the proper place in its time-line. In the case of even-driven simulators, the RTI will tell the federate how far. Clock-driven federates and event-driven federates can "play" together in the same federation.

The HLA provides three timing schemes for federates. A federate can be *time regulating*, *time constrained*, both *time regulating and time constrained*, or *neither time regulating nor time constrained*. A time regulating federate can pace the rest of the federation. The RTI will not grant time advance requests from the other federates until the regulating federate has caught up with them. A time constrained federate can be paced by the rest of the federation. The RTI will not grant its time advance requests until the rest of the federation has caught up with it. A federate that is both time regulating and time constrained can both pace the federation and be paced by it. And finally, a federate that is neither time regulating nor time constrained is not time-managed and is therefore unaffected by the time positions of the other federates. The RTI grants time advance requests from an unmanaged federate unconditionally and delivers events to it as they occur without regard to their time-stamps.

Time management is a major and powerful feature of the HLA. It allows federates to keep pace with the other federates without having to have complex custom synchronization features built into them.

Object Management

This is the RTI service, which facilitates data sharing within a federation. It allows federates to create new instances of objects in the federation and other federates to "discover" them. It allows data producing federates to publish objects and interactions and data consuming federates to subscribe to them. In order to comply with the HLA specifications, all federates within a federation must communicate with each other exclusively through the RTI Object Management services.

Data Distribution Management (DDM)

This service allows federates to associate objects with regions of interest and direct the RTI to filter subscribed events and deliver them to a given federate only if they are of interest. For example, a simulated tracking station may subscribe to some of the attributes of a spacecraft object, but wish to receive attribute update notifications only when the "spacecraft" comes within its cone of visibility. One could implement this type of filtering by defining a *routing space* for the federation based upon geographic parameters and relate it to that object. The federate simulating a tracking station would call the RTI's DDM services to define a region of interest within that routing space based upon its cone of visibility. The RTI would subsequently deliver attribute update notifications for that object to that federate only while the "spacecraft" is within that region of the routing space.

The R&D Study and Prototype

Purpose

In order to evaluate the suitability of the HLA as a technology for NASA space mission simulations, the CSOC Simulations Group conducted a study of the HLA and developed a simple prototype HLA-based space mission simulator. The CSOC Simulations Group and our customer at NASA Goddard Space Flight Center (GSFC) are interested in modeling overall space missions. This involves numerous, separate applications simulating the complex and multifarious aspects of a space mission and its numerous physical components. Examples of these components are spacecraft, on-board instruments, telemetry, tracking stations, ground systems, flight dynamics, and in the case of science missions, the natural objects and phenomena under study. We believe that many future missions will involve constellations and formations of spacecraft working together. The collaborative interactions of multiple spacecraft will add a new and complex, but exciting, dynamic to missions and modeling them early in their development cycle will be crucial to their ultimate successes. The HLA is possibly one of the technologies that could make these sophisticated new simulation systems realities. The study focused on the HLA and its potential in this area and this paper documents that study and the author's conclusions.

Methodology

We first read existing documentation and other literature on the HLA and attended training sessions provided by DMSO. This provided a fundamental knowledge base from which to work. We then formulated a basic concept of how an overall space mission simulation would work, the primary components involved, and how the components would interact. The HLA was to provide the basic architecture and framework for the simulation. In order to investigate the technology within the scope of a very small and inexpensive prototyping effort, the simulation had to be limited to only several component simulators and they had to be very simple. We developed the concept of operations or "ConOps" for the prototype and documented it in a slide presentation.

The next step after getting peer-level feedback from the ConOps presentation was to develop a list of high-level requirements for the prototype. These included minimum essential requirements and optional "nice to have" requirements to be implemented only if time permitted. We then searched through the free and open-source software community for free software components, which we could use and were delighted with what we found. We designed the prototype to make use of free software and of Government-owned software developed by the Flight Dynamics group within GSFC. We followed this with a fully object-oriented design for the prototype and documented it in the Unified Modeling Language (UML). Senior members of the technical staff reviewed the design at a formal design inspection. We also sent it to our NASA customer.

After receiving feedback from the design inspection, implementation began. Implementation went rather quickly, since the requirements and design were reasonably complete and stable. The exclusive use of free software components in the prototype allowed us to proceed without the delays and nuisances of licensing restrictions associated with proprietary software.

Platforms and Software Used

In order to reduce costs, the prototype was built using entirely free or in-house developed software and it binds to no proprietary software or libraries. A secondary goal of the prototyping effort was to validate the massive public library of free and open-source software as a source of usable components. The components we used performed very well and met all of our expectations.

Linux™

The prototype was developed on and currently runs on a standard Pentium II®-based desktop computer running the Linux[1] operating system. Linux (more appropriately called "GNU/Linux") is a free, mostly POSIX-compliant, and open-source Unix-like operating system that has gained enormous popularity in recent years. Due to its stability, the plethora of high-quality free software available for it, and its inherent upwards compatibility with Unix, Linux turned out to be an excellent development platform for this project. Linux and most of the software that runs on it is licensed under the GNU General Public License (GPL). The GPL allows for the free use, copying, modification, and redistribution of software, including its source code.

GSFC Flight Dynamics Programs

In order to generate simulated spacecraft orbital positions and to simulate Earth magnetic field data, the prototype made use of a set of programs previously developed by the Flight Dynamics group at GSFC. Because they are Government-owned, these programs were available to us without cost. These programs are written in Matlab® and required only modest modifications in order for us to incorporate them into the prototype.

Octave

Octave[2] is a free and open-source fourth generation (4GL) programming language and environment, which is mostly source-compatible with Matlab®. We used Octave to run the Flight Dynamics programs described above. They ran fine under Octave with only a few very minor modifications. Octave is publicly available and released under the GPL.

gnuplot

gnuplot is a free, but sophisticated, graphics program. We used it to generate all of the graphics.

DMSO Runtime Infrastructure (RTI)

The DMSO RTI is at the heart of the system. It is the HLA-compliant runtime infrastructure, which manages the simulation and handles all communications between applications. It consists of a library, which implements the HLA-defined application program interface (API) and a set of daemons, which run the simulation and handle communications. It is a certified and tested HLA RTI and is available on several platforms including Linux. DMSO provides and, for now at least, supports the software free of charge to U.S.-based organizations. The RTI will be discussed in detail in a later section of this paper.

[1] Linux™ is a trademark of Linus Torvalds.
[2] See http://www.che.wisc.edu/octave/.

The Prototype

The prototype space mission simulator simplistically models a rudimentary constellation of two Earth-orbiting spacecraft collecting science data. A "tracking station" monitors the positions of the two spacecraft in simulation time and a simulated Earth generates science data for their on-board instruments to collect. In its present form, this simulator does nothing that would be of direct use to any real mission or application; but it does illustrate the overall concept of how a useable space mission simulator might work and it demonstrates the role that HLA technology could play in such a system.

The prototype space mission simulator consists of four small, specialized simulators: a Spacecraft Controller, an Orbit Calculator, an Earth simulator, and a Tracking Station. These four simulators are all fully HLA-compliant and are completely separate applications. They run concurrently and separately, but not independently, of each other in individual processes and optionally on separate machines. As is required for HLA-compliance, the four simulators each define the data they share with the rest of the simulation in a Simulation Object Model (SOM) documented according to HLA specifications. Collectively, these SOMs make up the federation's FOM, which is also documented according to HLA specifications. These four simulators are federates and they communicate exclusively through the RTI to make up an HLA federation.

The FOM defines one HLA object class for the simulation, a spacecraft object; and the simulation creates two instances of it. One instance loosely represents the Landsat 7 spacecraft and the other the Terra spacecraft. This simplistically models a two-spacecraft constellation. We selected Landsat 7 and Terra because they follow similar orbits and their orbital elements are readily available. However, their orbital paths are really the only aspects of these missions that are meaningfully modeled by this prototype.

Internally, these simulators do little more than move data around, but all of their external interactions are handled by the RTI and they maintain full HLA-compliance. Again, the purpose of the prototype was not to realistically model a space mission, but to study and evaluate the HLA and to get an idea of how one might eventually use the HLA to build a useful space mission simulator in the near future.

The Spacecraft Controller

The Spacecraft Controller simulates the two spacecraft. It creates the two instances of spacecraft objects and registers them with the RTI. It also manages a simulated magnetometer science instrument on-board the "Landsat 7" spacecraft.[1] This instrument is implemented simply as a data structure to which magnetic field data are written. The spacecraft controller then writes these data out to a temporary file from which *gnuplot* reads them and graphs them on a simple line chart for the user. Although the Spacecraft Controller creates and maintains the spacecraft, it does not update any of their attributes. In fact, it relinquishes ownership of their positional attributes to the Orbit Calculator federate and ownership of the magnetometer science instrument attribute to the Earth federate. It retains ownership only of the inherited *privilegeToDeleteObject* attribute, which it needs in order to delete the objects from the federation when the simulation is done. In order to keep pace with the federation, the Spacecraft Controller makes use of the RTI's time management services as a time regulating and time constrained federate.

[1] The real Landsat 7 spacecraft does not have a magnetometer as a science instrument.

The Orbit Calculator

The Orbit Calculator computes the orbital paths of the two simulated spacecraft based on the two-line orbital elements of the real Landsat 7 and Terra spacecraft. For each of the spacecraft, it computes nadir geographic ground coordinates as a function of simulation time. The Orbit Calculator acquires ownership of the spacecraft objects' positional attributes from the Spacecraft Controller, publishes them, and updates them with the ground positions that it has read for them with each advance of its clock. The Orbit Calculator is a time regulating and time constrained federate. In fact, its sets the pace of the federation by retrieving nadir ground positions with each advance of its clock. The time step of its internal clock is the interval between each data point in its precomputed orbital position file.

In order to generate nadir ground positions for the simulated spacecraft, the Orbit Calculator reads a file containing precomputed positions for the two spacecraft for the interval of simulation time. The GSFC Flight Dynamics group has developed programs that perform these calculations for any satellite using its two-line orbital elements and a specified time interval. The Landsat 7 and Terra spacecraft's two-line orbital elements are publicly available. We obtained the Matlab® source files for these programs from the Flight Dynamics group and made some modifications. Running under Octave, the programs generate the files containing the positions of the two spacecraft for the desired time interval and the Orbit Calculator later reads them.

The Tracking Station

The Tracking Station simulator subscribes to the spacecraft objects' positional attributes, but does not update them. It reads the attributes each time that the RTI notifies it of updates and writes the data to a temporary external file. A separate *gnuplot* process reads the file and displays the positions of the two spacecraft on a map of the Earth as a function of simulation time. The result is a simple graphic that depicts the paths of the two spacecraft over the Earth during the simulation. Since the Tracking Station does not publish data, no other simulators are dependent upon it. Therefore, the Tracking Station federate is time constrained only. The federation paces it, but it cannot pace the federation.

The Earth Simulator

The Earth simulator produces simulated earth sciences data that a simulated instrument on-board the spacecraft can sample. In this case, it generates Earth magnetic field data for the "magnetometer" on-board the simulated Landsat 7 to measure.

Like the Orbit Calculator, the Earth simulator reads an external file containing precomputed data. A program, also obtained from the GSFC Flight Dynamics group, computes magnetic field intensities as three-component vectors for a set of positions on the Earth and writes them to a file. These positions match those for the paths of the spacecraft computed for the Orbit Calculator. We run the magnetic field program under Octave and generate the data file prior to running the Earth simulator.

The Data Flow

The Spacecraft Controller, Orbit Calculator, Tracking Station, and Earth Simulator are all HLA federates. Together, they make up an HLA federation. Each federate runs in its own process as a separate and stand-alone application. The federates run together on multiple machines as parts of a distributed computing system and exchange data exclusively through the RTI.

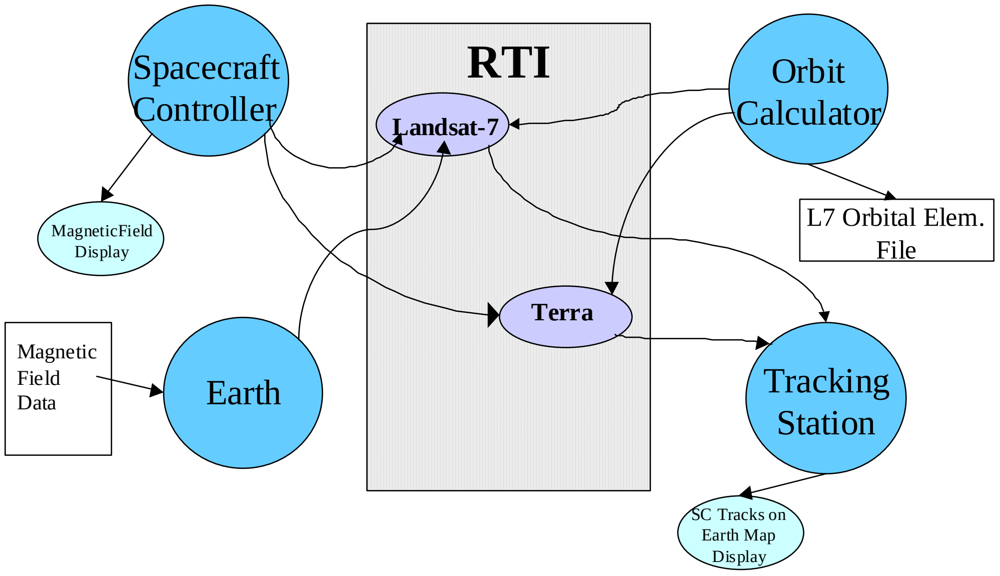
The user starts the four simulator applications, hereafter referred to as “federates”, and the *rtiexec* daemon separately on their respective machines from the command lines. The first federate to come up creates the federation and causes the *rtiexec* daemon to spawn the *fedex* daemon. As each federate comes up, it registers with the RTI and “joins” the federation. All of the federates subscribe to the *Start_Time_Msg* and *SimulationEnds* interactions. The Orbit Calculator federate publishes the positional attributes of the spacecraft object class and the Earth federate publishes the magnetometer instrument attribute. The Tracking Station federate subscribes to the positional attributes of the spacecraft object class and the Spacecraft Controller subscribes to the magnetometer attribute. After successfully joining the federation, the Spacecraft Controller creates the “Landsat 7” and “Terra” instances of the spacecraft object class and registers them with the RTI.

The data flow begins with the Orbit Calculator. This federate opens the precomputed orbital positions files for the two spacecraft and reads the simulation start time and the starting positions of the two “spacecraft.” It sets its internal clock to the simulation start time, packs the value into the *Start_Time_Msg* interaction, and sends it. Upon receipt of this interaction, each of the other federates sets its internal clock to the simulation start time and requests a time advance grant from the RTI to that time. If a federate joins the federation late and misses the *Start_Time_Msg* interaction, it queries the RTI for the current federation time and sets its internal clock accordingly. This scheme allows the federates to self-synchronize with the rest of the federation. The Orbit Calculator acquires ownership of the two spacecraft object’s positional attributes and updates them with the positions that it has read from the external file. It then reads the next simulation time and corresponding set of positions and requests a time advance from the RTI for that time. Once the RTI grants the request, the Orbit Calculator advances its internal clock and again updates the positional attributes of the spacecraft objects. This cycle repeats until all of the data in the position files have been read. Since the Orbit Calculator is a time regulating federate, it paces the rest of the federation. Since it is also a time constrained federate, the RTI will not grant its time advance requests until the rest of the federation has caught up with it. When the Orbit Calculator has read the last time and position value set from the position files, it sends the *SimulationEnds* interaction to the federation. It then resigns from the federation and terminates. When the other federates receive this interaction, they also resign from the federation and terminate. The simulation is done when the last federate has terminated.

Each time that the Orbit Calculator updates the attributes of one of the spacecraft object instances, the RTI sends a notification to the federates which have subscribed to them. The Tracking Station federate reads the positional attributes for both spacecraft each time the RTI informs it of an update. As it receives them, it writes the positional data to a temporary external file and a separate process repeatedly evokes *gnuplot* to read this file. *gnuplot* plots the positions on a map of the Earth and displays them on the screen. The result is a continuously updated map of the Earth with the tracks of the two spacecraft displayed on it in simulation time.

Like the Orbit Calculator, the Earth simulator reads an external data file generated by another Flight Dynamics program running under Octave. This file contains Earth magnetic field intensity data that correspond to the spacecraft’s nadir pointing ground positions in simulation time. There is an exact one-to-one temporal correspondence between the positional data points read by the Orbit Calculator and the magnetic field data points read by the Earth federate. These magnetic field intensity values simulate science data that would be measured by a magnetometer instrument on-board a spacecraft. With each advance of its internal clock, the Earth federate updates the "magnetometer" attribute of the "Landsat 7" object with a value read from the magnetic field file. Like the other federates, it advances its clock only when the RTI grants it permission to do so. The Earth federate is both time regulating and time constrained, so it will not update the spacecraft's attribute with the next data point until the RTI sends it a time advance grant. Therefore, the federation cannot get ahead of it on its time-line and it cannot get ahead of the federation.

In order to simulate an on-board instrument collecting science data, the Spacecraft Controller subscribes to the magnetometer attribute of the spacecraft objects. Each time the Earth federate updates the attribute, the RTI informs the Spacecraft Controller and the Spacecraft Controller reflects (reads) the attribute and writes its value to a temporary external file. A separate process evokes *gnuplot* to read this file and display a line graph of the three magnetic field intensity vector components as a function of simulation time. Through its time management services, the RTI makes sure that all of this happens while the federates are at the same point in their simulation time-lines. Thus even though their respective data are being generated by separate applications, the magnetic field intensity graph temporally matches the display of the spacecraft ground tracks.



• **Figure 1 A diagram of the data flow within the prototype Space Mission Simulator**

We normally run the Space Mission simulator on two separate Linux machines, typically with the Spacecraft Controller and the Tracking Station on one and the Orbit Calculator and Earth simulator on the other. The RTI transparently handles the network connections and data communications. From the point of view of the federation, it does not matter which federates are running on which machines. Theoretically, one could run an entire federation on a single machine; however, we found that two or more of these applications running on a single Pentium II®-class processor taxes the system resources beyond acceptable limits.

Software Design

The Space Mission simulator design follows a strict object-oriented paradigm and is documented using UML syntax. Each federate is implemented as a class, which is derived from a higher-level abstract base class. The higher level abstract base implements the HLA. The Spacecraft objects and the interactions are also implemented as classes and are derived from higher-level abstract base classes. To the greatest degree possible, the HLA and its interface to the RTI are implemented in the base classes and the simulator-specific functionality in the derived classes.

The federates are written entirely in C++. Every effort was made to avoid platform or vendor-specific coding constructs and system calls. Although developed on a Linux system, they conform to ANSI and POSIX standards and should be portable to other platforms, including Windows NT[1]. Porting them to Unix systems, especially if one uses the widely available GNU C++ compiler (*gcc*), should be easy. The programs, which generate orbital positions and magnetic field intensities, are written in Octave. Octave is available only on Unix and Linux; however, because it is source-compatible with Matlab®, these programs should be highly portable also. Versions of *gnuplot* are available for a wide number of platforms.

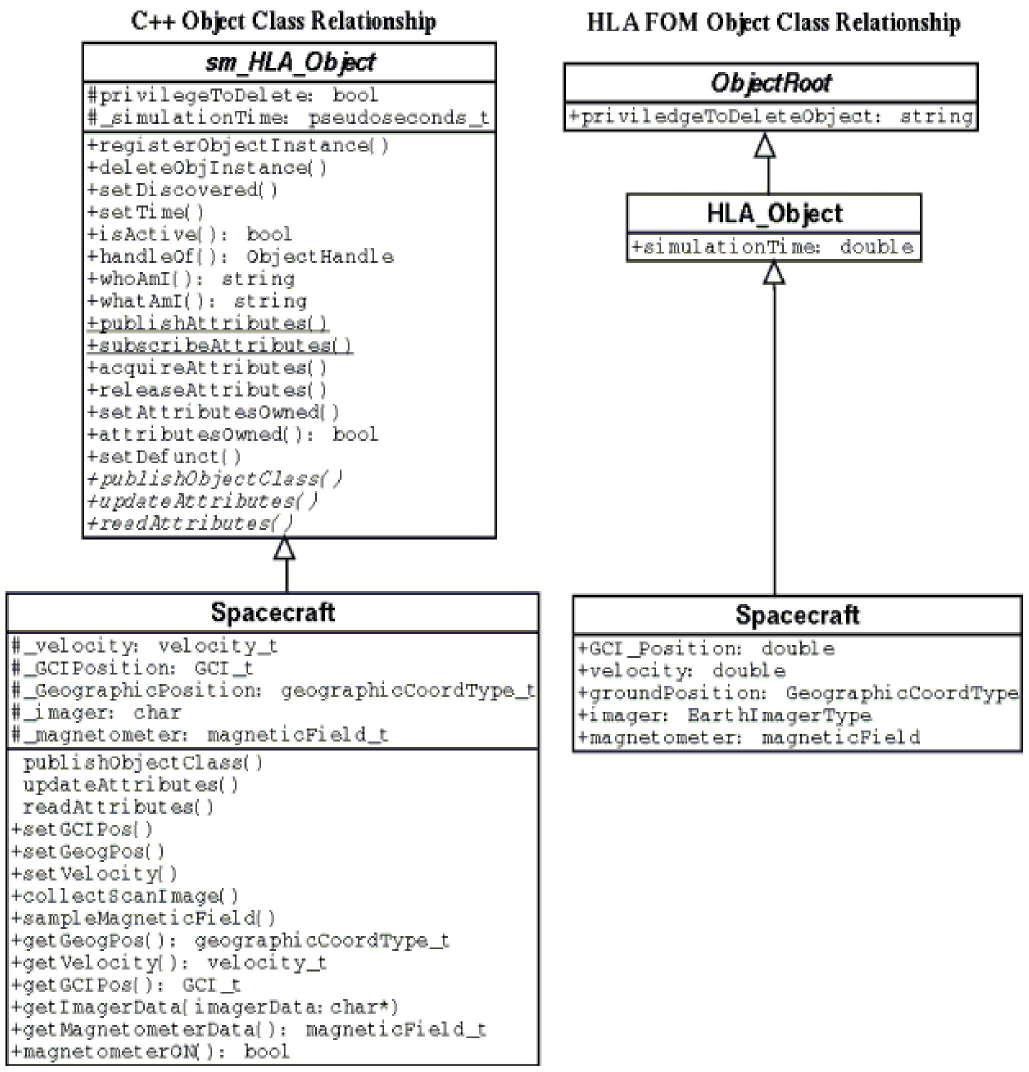
The Base Object Class

The design for the prototype includes an abstract base class called *sm_HLA_Object* and requires that every HLA object class defined in the FOM have a corresponding C++ class derived from this base class. The derived C++ classes inherit RTI interfacing functions from the base class and extend it by including attributes that precisely match those defined in the associated FOM object class. Whenever a federate creates or discovers an instance of a FOM object, it mirrors it by creating a local instance of the corresponding C++ class. The base class does not contain publishable attributes of its own[2], since they are specific to the particular type of object. These are left to the object-specific derived classes. But it does define member functions to create, discover, and destroy instances of the object class and other member functions to publish, subscribe, update, and reflect (read) the derived object's publishable attributes. Some of the base class's member functions are purely virtual since they depend upon object-specific attributes.

This prototype's FOM defines only one object class, a *Spacecraft* object. The prototype's C++ code contains a corresponding class, *Spacecraft*, derived from *sm_HLA_Object*. A federate or group of federates may create and register any number of instances of *Spacecraft*. The Spacecraft Controller creates two such instances, one it calls "Landsat 7" and the other "Terra". The two *Spacecraft* instances are identical in form, but the RTI knows them by different names and their attributes contain different values. This ability to create any number of instances of an object class is a major feature of the HLA. In the prototype, the other three federates subscribe to the *Spacecraft* class and whenever they discover a new instance of it, they create corresponding local instances of the C++ version of the class. When a federate wishes to acquire or relinquish attribute ownership, update, or read the attributes of one of the HLA object instances, it simply calls the associated public member function in the C++ object. The member function in turn calls the appropriate functions in the RTI ambassador to perform the operation.

[1] Windows NT® is a trademark of Microsoft Corporation.

[2] Except for a counterpart to the HLA-defined *privilegeToDeleteObject* attribute inherited from the predefined HLA Object Root class.



- **Figure 2 The above UML class diagrams show the class relationship between the HLA_Object base class and the derived Spacecraft class. The diagram on the left is the C++ definition and the corresponding HLA FOM definition of the class is on the right. Some less important attributes have been omitted from the C++ diagram. "Landsat 7" and "Terra" are instances of the *Spacecraft* object class.**

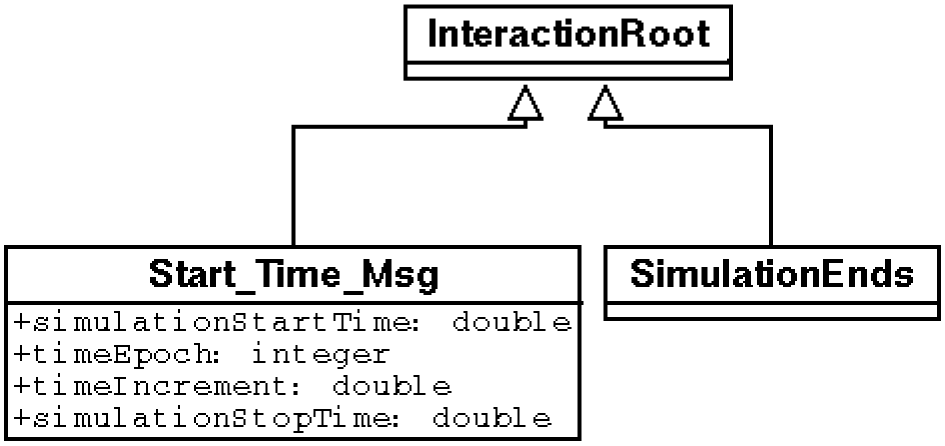
Of course in a realistic simulation, the *Spacecraft* class would have far more attributes. Since the prototype only models the positions of spacecraft and an on-board magnetometer science instrument, it only contains attributes for them. One will notice an *imager* attribute as well. This is for a simulated imager science instrument, which was not implemented in the current prototype. In the future, one could be more specific as to the types of spacecraft modeled and carry the class hierarchy to more levels. Since both C++ and the HLA FOM standards fully support class inheritance, one could model specific types of spacecraft by deriving additional classes from the *Spacecraft* class. One could reuse the *sm_HLA_Object* as a base class for creating nearly any other type of HLA object as well. It is by no means limited to parenting spacecraft-type objects.

The Base Interaction Class

The design of the prototype implements HLA interactions in much the same way as HLA objects. There is an abstract base class, *sm_HLA_Interaction*, from which C++ versions of all of the interactions defined in the FOM are derived. The base class declares and implements member functions to create, publish, subscribe to, and send interactions. The derived C++ interactions extend the base class for their data-specific needs and inherit its member functions. As with objects, federates publish and subscribe to interactions. When a federate sends an interaction, the RTI delivers it to all subscribing federates based on the interaction's time-stamp. The sending federate creates an instance of the desired interaction class, assigns values to its parameters, and calls a member function to send it. The member function in turn calls the appropriate functions in the RTI ambassador to marshal the parameters and send the interaction to the federation. Unlike HLA objects, interactions do not persist after the RTI sends them.

The prototype design derives two subclasses from the *sm_HLA_Interaction* base class, the *StartTimeMsg* and the *SimulationEnds* interactions. As the names imply, the *StartTimeMsg* informs the federates of the simulation start time and the *SimulationEnds* informs them of the end of the simulation. All federates subscribe to both of these interactions. The Orbit Calculator federate reads the first data point from one of the spacecraft positions files along with its time. It creates an instance of the *StartTimeMsg* interaction, packs the time and a time increment into it and sends it out. When a federate receives the interaction it sets its internal clock to the start time and the increment and requests a time advance from the RTI to that point in time. From that point on, it sets its current internal time and requests time advances from the RTI to the current time plus the time increment. This along with the RTI's time management keeps the federates synchronized. Finally, when the Orbit Calculator reads the last data point from the positions file, it creates and sends out a *SimulationEnds* interaction. When the federates receive this interaction, they finish what they are doing, tidy up, and terminate.

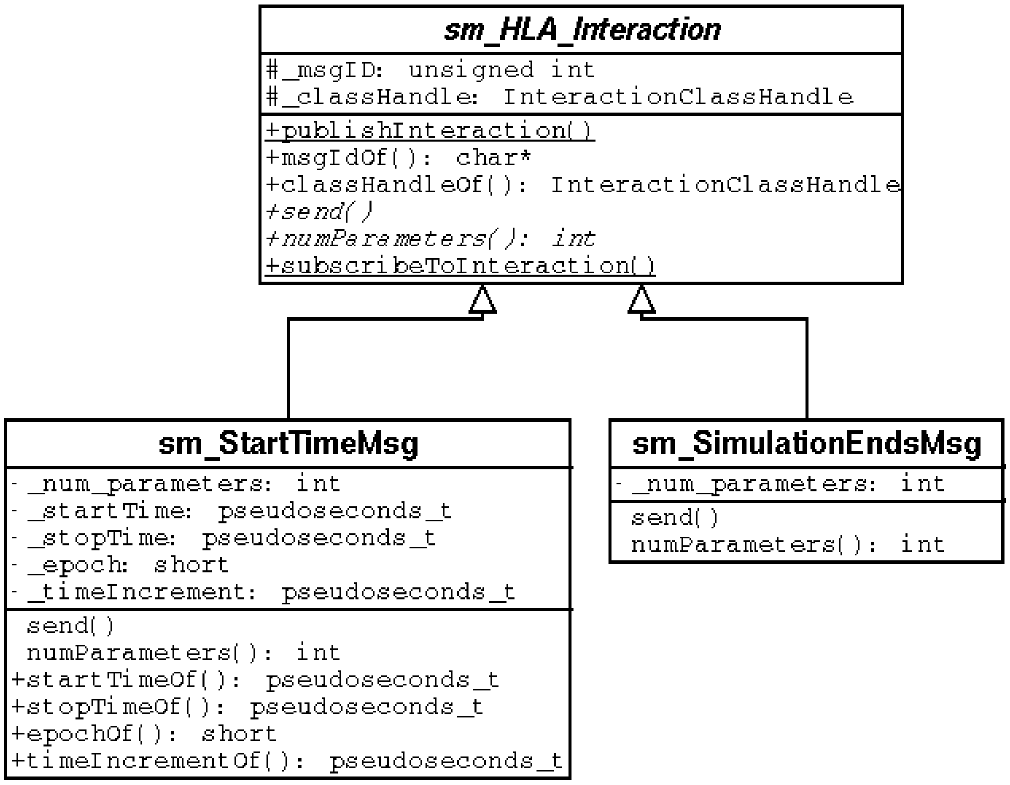
HLA FOM Interaction Class Relationship



•

Figure 3 **The above UML class diagram shows the class relationship between the FOM definition of the HLA_Interaction base class and the two derived interaction classes.**

C++ Interaction Class Relationship


sm_HLA_Interaction
#_msgID: unsigned int
#_classHandle: InteractionClassHandle
+publishInteraction()
+msgIdOf(): char*
+classHandleOf(): InteractionClassHandle
+send()
+numParameters(): int
+subscribeToInteraction()
sm_StartTimeMsg
- _num_parameters: int
- _startTime: pseudoseconds_t
- _stopTime: pseudoseconds_t
- _epoch: short
- _timeIncrement: pseudoseconds_t
send()
numParameters(): int
+startTimeOf(): pseudoseconds_t
+stopTimeOf(): pseudoseconds_t
+epochOf(): short
+timeIncrementOf(): pseudoseconds_t
sm_SimulationEndsMsg
- _num_parameters: int
send()
numParameters(): int

•

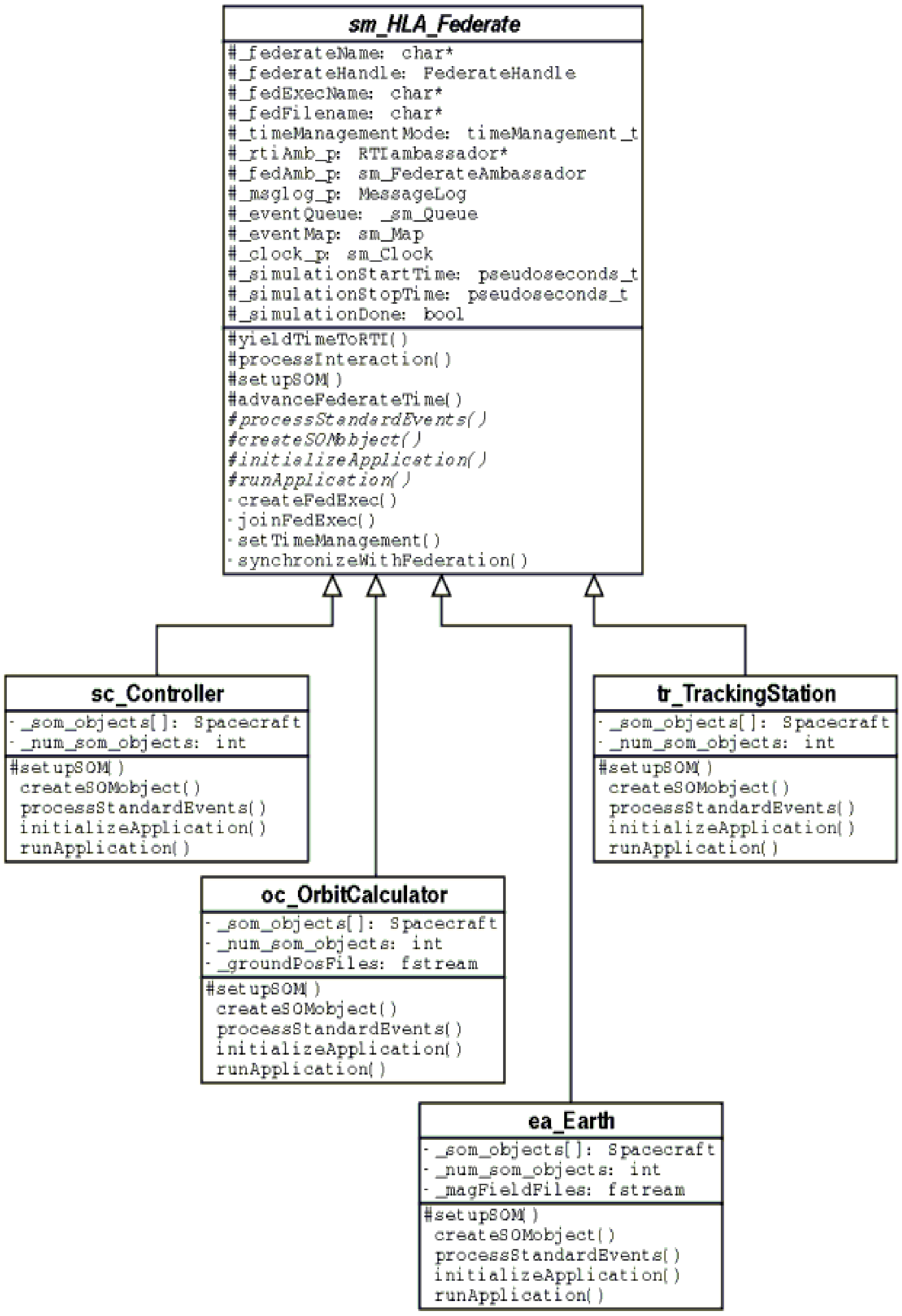
Figure 4 The above UML class diagram shows the class relationship between the C++ definition of the HLA_Interaction base class and the two derived interaction classes. Some less important attributes have been omitted from the diagram.

One could reuse the *sm_HLA_Interaction* as a base class for creating nearly any other type of HLA interaction as well. As with the base object class, it is not limited to parenting just these interactions.

The Base Federate Class

Following the object-oriented paradigm, all federates in the prototype are derived from a base federate class, *sm_HLA_Federate*. As part of a federation, any HLA-compliant federate must be able to create a federation execution, join a federation, request time management services, synchronize with the federation, publish and subscribe to objects and interactions, and resign from the federation. The *sm_HLA_Federate* abstract base class provides functions for doing these things and it is designed to work with the object and interaction classes described above. It uses the methods they provide for publishing and subscribing to them and for reading and updating their attributes and parameters. This design separates the HLA-specific functionality in a federate from its own application-specific functionality and makes creating new federates much easier, since the HLA interfacing is prepackaged for them and they need only implement their own specific processing. All of the protected member functions in *sm_HLA_Federate* are declared virtual so that a derived class can override them with a version of its own if necessary. Some of the application-dependent functions are pure virtual functions, so a derived class must implement them locally. The base class maintains, among other things, instances of the federate ambassador and RTI ambassador, a message log, and an internal clock within its protected data. The derived federate classes inherit these, along with the RTI interfacing member functions. As with the object and interaction abstract base classes described earlier, the *sm_HLA_Federate* class could be used in other unrelated applications.

HLA Federate Base Class

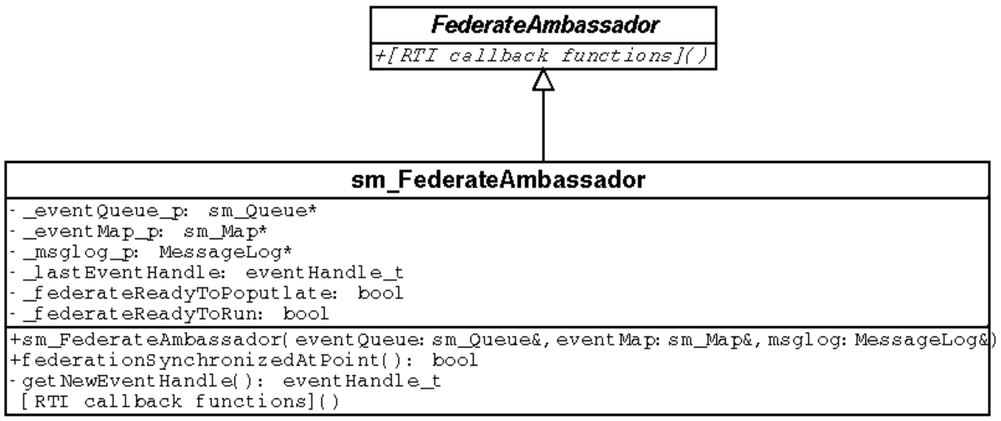


- **Figure 5 The diagram above depicts the dependency relationship between the federate base class and the derived application-specific federate classes.**

The Federate Ambassador Class

Every HLA federate must implement a federate ambassador. The federate ambassador is a class containing a number of member functions that define the part of the HLA RTI interface that the RTI software evokes as callbacks to send information to the federate. An application creates the federate ambassador by deriving its own class from a base class provided with the RTI and implementing its member functions locally.

In the prototype system, the federates all share the same local implementation of the federate ambassador. The derived local federate ambassador class includes a private section that contains several attributes (i.e., *attributes* in a C++ sense): pointers to an event queue, a map, and a message log object. A class constructor has also been added to assign values to these attributes. The federate creates and maintains the queue, map, and message log objects. Pointers to these objects are passed to the federate ambassador as arguments to its constructor. Whenever a member function is called, it places a description of the event on the event queue and stores any associated data (such as reflected HLA object attributes) in the event map. Since the queue and map are pointers to objects maintained by the federate, the federate has visibility to them. As the RTI generates events and communicates them to the federate through its federate ambassador, the federate handles them in turn inside an event loop. This design approach avoids the undesirable need for global data structures and maintains the object-oriented principle of data hiding.


FederateAmbassador
+[RTI callback functions]()
sm_FederateAmbassador
- _eventQueue_p: sm_Queue*
- _eventMap_p: sm_Map*
- _msglog_p: MessageLog*
- _lastEventHandle: eventHandle_t
- _federateReadyToPoputlate: bool
- _federateReadyToRun: bool
+sm_FederateAmbassador(eventQueue: sm_Queue&, eventMap: sm_Map&, msglog: MessageLog&)
+federationSynchronizedAtPoint(): bool
- getNewEventHandle(): eventHandle_t
[RTI callback functions]()

• **Figure 6** **The diagram above shows the dependency relationship between the base federate ambassador class provided with the RTI and the local, application-specific derived class. The member functions declared in the base class are too numerous to depict in the diagram. The derived class diagram only shows the functions and attributes added for the local federate ambassador.**

Code Reuse

In the prototype, message logging and exceptions are provided by code reused from an earlier GSFC project. The event queue and the event map classes used by the local federate ambassador are instantiated from the C++ Standard Template Library (STL).

The HLA object, interaction, federate, and federate ambassador abstract base classes and their supporting classes are compiled into a Linux shared object library. This library should be fully reusable by future HLA-based systems.

Evaluation of the HLA as a Technology

What the HLA Is

The HLA is an architecture for simulations. It is a current DoD and an upcoming wider industry standard (IEEE-1516). It consists of a set of rules and defines a standard interface to a runtime infrastructure (RTI). The RTI facilitates communication among individual simulator applications, provides time and data management services, and manages the overall simulation. HLA simulations are designed to run as distributed systems over a network.

What the HLA Is Not

The HLA is not, in itself, a simulator or a modeling tool. It is not a rapid application development system for simulations. It provides no data display capabilities or user interface. Although intended to make integrating disparate and remote simulators more practical, it will not make "plug and play" a reality. Even within the context of an HLA-compliant simulation, the persons responsible for the various simulators must still negotiate common data types in order to make their systems compatible. If the simulators are fully HLA-compliant, the mechanisms by which they may share data are already established, but the individual applications must still have knowledge of those specific data that they share. It is important to reiterate that the HLA is not a simulator. It is an architecture around which one may design and integrate application-specific simulators.

What the HLA Is Good For

The HLA is especially well suited as basis for game-like simulations in which a little "universe" is being simulated with multiple actors coming and going and interacting. This is not surprising, since the HLA was originally designed for war-games. It also has potential as a framework for integrating numerous separate simulators into a larger distributed simulation system. The fact that HLA-based simulators are designed around data classes from which any number of instances may be created give them great potential for easy expandability. The HLA's inherent distributed nature should also allow for essentially unlimited scalability.

What the HLA Is Not Good For

The HLA is probably not a good architecture for developing simulations, which are not game-like or are not composed of multiple separate and interacting parts. If testing and validating a component's external communication mechanism is part of the simulation, the HLA is not going to help. Simulators based on a single point-to-point stimulus and response model are also not good candidates for implementation around the HLA, since they generally do not share persistent data and do not manage multiple instances of simulated objects. In other words, systems that do not lend themselves to modeling as groups of interacting objects are not good fits for the HLA. Simulators of on-board spacecraft systems are generally required to validate the physical interfaces among their various components. Therefore, the HLA is probably not a good architecture for such systems either. The DMSO RTI uses CORBA internally and thus, incurs that technology's know performance overhead and is probably not well suited for most real-time applications.

What the HLA Will Do

The RTI will handle the communications and data sharing among remote applications provided that they all conform to the HLA paradigm and will handle the connectivity between applications running on different platforms. It will also provide time management and synchronization services. The RTI is a sophisticated and capable infrastructure, which does a lot of the generalized work not specific to given applications. HLA-compliance will definitely make it much easier and less costly to integrate disparate simulators into larger simulations, provided that the individual simulators are all truly and fully HLA-compliant.

What the HLA Will Not Do

The HLA will not do your simulation for you or increase the fidelity of your simulators. It will not generate or display your data or provide you with a user interface. It is up to the applications to provide such features. Since few COTS products are currently HLA-compliant, the HLA will not make it easy to integrate them into other systems. However, it should be possible to build custom interfacing software for the COTS products that would allow them to participate in HLA-based simulations.

Observations on the RTI

The prototype uses the Linux version of the DMSO RTI ver. 1.3NG. This RTI software does appear to implement the HLA RTI specification faithfully and completely. The only significant deviation from the specification that we observed in the DMSO RTI is the *tick()* function, which they have added to the RTI ambassador. The *tick()* function is not defined in the HLA specification. The DMSO RTI requires that a federate call it frequently in order to yield processor time to the LRC. We observed that the frequency and timing of *tick()* calls is very important. Too few tick calls and the LRC fails to deliver callbacks to the federate. Too many *tick()* calls in the wrong places and the federate never performs its specific processing. Although not really a problem, some experimentation was required in order to find the right places to call this function. Having to deal with *tick()* is a little annoying and it would be nice if a future version of the RTI could dispense with it.

The DMSO RTI is very sensitive to the host system's network configuration. Small misconfigurations that do not affect common network applications such as telnet, ftp, or Web browsers can affect the RTI and cause it not to work. More problematically, the RTI generally does not give useful error messages in these situations. Again, this is not really a problem with the RTI, but it would be nice if it gave useful enough error messages so that a user could diagnose the problem without outside assistance.

We found that the federation execution needs to run either as a distributed system or on higher-end hardware. The RTI software consumes a lot of system resources. In the case of the prototype, it could only run two federates at a time on the same PC without an unacceptable performance degradation. We solved this by using two machines. Of course, the downside of running as a distributed system is that the load on the network affects performance. Therefore, it is probably best to run HLA federation executions over closed networks. The RTI will run over the Internet, but performance will vary dramatically based on the network traffic. In order to collect meaningful performance data, we would have needed exclusive use of a closed network and this was not available for the prototype.

Overall, the Linux version of the DMSO RTI works as advertised. It is a fully usable, "industrial strength" product and the technical support provided by SAIC, under contract to DMSO, is quite good. Best of all, their RTI runs on numerous platforms.

Considerations

There are some facts that one should take into consideration before adopting the HLA as the core architecture for a system. The HLA comes with a fairly steep learning curve. The effort required for a system designer or programmer to become familiar enough with the technology to use it effectively is tantamount to that required to learn an entirely new programming language. At present, documentation on the HLA is sparse and good code examples are lacking. DMSO does provide some sample programs with their RTI, but they are not well documented.

Since most COTS products and legacy systems are not yet HLA-compliant and may never be, one would need to develop custom interfacing applications in order to integrate them into a simulation system. Although quite doable in most cases, the tasks would not be trivial. The HLA facilitates the reuse of preexisting simulators, but not as "plug and play" modules. The preexisting application's SOM must still be reconciled with the new simulation system's FOM and this will nearly always require changes to one or the other. These factors must be taken into account in any project plan so that adequate time and resources are available to address them.

Although we could not compile quantitative performance statistics during the prototyping effort, the current RTI does not appear to be fast enough for applications with hard, real-time performance requirements or for high-performance computing. But since game-like simulations run in their own simulation time, not by the wall clock, this should generally not be a problem. Hopefully in the future, commercial vendors will provide faster RTIs with better error messages and more documentation.

It is important to keep in mind that the HLA is still a maturing technology. As of this writing, the IEEE-1516 specification is relatively stable, but not finalized. Currently, the DMSO RTI is the only fully compliant and fully tested RTI available for most platforms. The Java and CORBA IDL interfaces to the RTI are available only on Windows NT® and Solaris[1]. It would be very nice if DMSO would port them to all of the platforms that they support, especially to Linux. Unfortunately, DMSO does not currently publish the source code for the RTI. In the absence of really good error messages and documentation, the lack of access to the RTI source code is an impediment to the successful implementation and testing of HLA-based systems and possibly to the HLA's wider acceptance. Although the on-line technical support provided by SAIC is quite good and so far free, access to the source code would make it much easier for application developers to diagnose problems. One huge advantage of using open-source software is that the source code is generally available. In the case of software licensed under the GPL, it is always available.

[1] Solaris® is a version of Unix and a trademark of Sun Microsystems.

The HLA and NASA

Although the missions of NASA and the DoD are fundamentally different, their technical needs often overlap. There is the potential for future collaborations with the DoD and software reuse. Budgetary realities will likely force NASA projects to rely more heavily on simulators for mission planning, testing, and training, so there could be a role for the HLA.

Future space missions will often involve constellations or formations of spacecraft working in unison. In order for simulators to model such missions, they will need to simulate multiple instances of spacecraft and ground stations and possibly multiple instances of natural objects in the space environment. They will also need to integrate different simulators together, usually as a distributed system running on multiple platforms. In the case of missions that rely on groups of spacecraft flying in formation, the simulators will need to simulate interactions between multiple spacecraft and ground stations and the natural environment. The designers of such simulations should seriously consider the HLA as a core technology upon which their systems could be based.

Space mission simulations will usually require the services of COTS products such as *STK*[1] or *FreeFlyer*[2] to provide mission and flight modeling and GOTS products such as the SIMSS to generate telemetry. These applications are not presently HLA-compliant and would require custom interfacing applications in order to participate in HLA-based simulations and work with custom simulators.

The author sees two particular areas of interest to NASA where the HLA has potential, space missions involving formations of spacecraft and the modeling of natural systems. Of course, there may be many more.

Formation Flying

For the purpose of this discussion, formation flying is here defined as a group of two or more spacecraft whose orbital trajectories are designed such that the relative positions of the spacecraft form a specific geometry, at least for a moment. The spacecraft are working together for some collaborative purpose and they communicate with each other either directly or indirectly through an intermediate ground station.

Examples of missions using formations of spacecraft are science missions where a three-dimensional sample of data is desired or where one needs data to be collected from multiple vantage points at the same moment. There is also the exciting concept of a formation of spacecraft each carrying telescopes and using interferometery to form, what is in affect, a single, much larger instrument. There would be multiple instances of interacting spacecraft, ground stations, natural objects, and even human controllers to simulate. These could be modeled as objects and thus, the HLA could provide a sound framework for large-scale simulations of such missions.

One possible design for such a simulation system would involve multiple SIMSS-based spacecraft generating telemetry, a COTS product providing the flight dynamics computations and graphical display, other custom simulators modeling the natural objects under study and generating science data, and still others modeling ground stations.

[1] STK® is a trademark of Analytical Graphics, Inc. (AGI).
[2] FreeFlyer® is a trademark of AI Solutions, Inc.

Natural Systems

Scientific studies of natural systems such as those of the Earth or of other planets often require models. Natural systems are immensely complex and involve innumerable smaller systems. For example, an earth sciences mission might require a collection of simulators. One simulator would model the oceans, both its physical and organic components. Another the atmosphere. Still others would model the magnetosphere, geological phenomena, and anthropic factors. These simulators would generate data for the "spacecraft" to collect. Since these phenomena are interdependent, dynamic and sometimes ephemeral, they could be modeled as collections of interacting objects. The ability of HLA-based simulations to create any number of instances of objects and to run as a highly modular distributed system could help make such complex simulations more possible and scalable.

The SIMSS

The HLA prototyping effort did not involve the current SIMSS spacecraft components. The current SIMSS simulates telemetry generated by a spacecraft and transmits that telemetry through either IP sockets or through an application specific interface. It is primarily a test tool and not used in mission modeling. The HLA has no application to the SIMSS in its present role or within its present user domain. However, telemetry generation would be a critical part of any large-scale, overall space mission simulation. The author proposes expanding the SIMSS such that it can participate as one component in a larger simulation and provide that crucial function.

The SIMSS is not HLA-compliant. In order for it to "play" in an HLA-based simulation, it would have to communicate through an interfacing application. This application would communicate with the SIMSS through IP sockets and with the rest of the simulation through the RTI. The SIMSS and its HLA interfacing module would together make up an HLA federate. The SIMSS would generate the necessary telemetry and send it through IP sockets to the interfacing application. The interfacing application would receive the telemetry stream through its IP sockets, copy it into HLA interactions, and send them to the federation. The spacecraft objects in the mission simulation would contain many attributes of which the telemetry stream would be one. Other simulators, including COTS tools operating through their own interfacing applications would run as federates and would take ownership of and update other attributes within the spacecraft objects. Given the complexity of spacecraft, there could be a great many attributes owned and updated by a great many different simulators.

This collaboration of many simulators, including some generating science data, could potentially produce a very high-fidelity simulation of a space mission. A major difficulty in developing such simulations is that the various simulators are separate applications and run on different platforms. This difficulty does not arise in HLA-based simulations, since the RTI handles all of the communications and inter-platform connectivity. Of course, the task of designing SOMs for each of the simulators and integrating them together into a single FOM remains.

Summary and Conclusions

The HLA is a current DoD and an upcoming IEEE standard architecture for simulations and modeling. It is not in itself a simulator, but does provide a framework for collaboratively integrating groups of disparate and remote simulators into larger simulation systems. The RTI software specified by the HLA handles the generic communication, connectivity, synchronization, and data visibility issues common to all distributed simulators and relieves them of the burden of managing the overall system infrastructure. The HLA is well defined, tested, and provides a welcome standard for large-scale, distributed simulation systems. It is best suited for game-like simulations where a little "universe" is being modeled and other types of simulations that involve multiple instances of interacting objects.

The HLA is probably not an appropriate architecture for simulations that just generate data for some single, external system or for simulations that cannot define their problem space as a collection of interacting objects. For systems such as these, the HLA would just be an unneeded extra layer and would provide little or no additional value. Simulators required to send data through a specific type of interface, such as IP or through a hardware bus are also not suited for the HLA, since bypassing the RTI for external communications violates one of the primary rules of the HLA. This is not to say that simulators such as these could not be formed into components of larger HLA-based simulations. They certainly could be.

The HLA could find a role in NASA as a core technology in certain types of space mission simulations, especially those involving multiple spacecraft and ground stations. It could also serve as the basis for the scientific modeling of natural systems. The HLA is a sophisticated and viable technology for some types of simulations, but it is important to reiterate that the HLA is not a simulator or a rapid development tool. It is a standard framework for integrating simulators. The HLA will not increase the fidelity of simulators, but its modular and distributed design will make it easier to build higher-fidelity simulators. In order to benefit from it, simulation systems must be designed from the beginning for HLA-compliance.

The HLA-based prototype space mission simulator is far too simplistic to be of use in any real mission, but that was not its purpose. Its purpose was to explore the HLA as a technology and evaluate its viability for NASA simulations. It also had a secondary purpose of demonstrating the viability of free, open-source software such as Linux. The author's conclusion on this is that much of the open-source software is as good or better than many commercial products, could be and should be used in operational systems. A side benefit resulting from the software development of the prototype is a set of reusable software components for implementing HLA-compliance.

The HLA is a solid technology and should have a place at NASA. It is well suited for the purpose for which it was designed. Like any technology, it is appropriate for some types of applications, but not for others.

Glossary

abstract base class	A class type provided by some object-oriented programming languages (e.g., C++), which contains at least one pure virtual function. One may only derive child classes from an abstract base class and may not create object instances directly from it.
ANSI	American National Standards Institute. An organization that defines standards and conventions for among other things, compilers and operating systems.
API	Application Program Interface. The set of interfaces to software functions provided by a library or other external software component.
attribute	An individual data item contained within an HLA FOM object. Also, data items contained within a C++ (or other OOP language) class.
base class	A class defined in the HLA FOM or a class defined in an object-oriented programming language (e.g., C++) from which other more specialized classes are derived with inheritance of functions and attributes.
COTS	Commercial Off-The-Shelf. This term usually refers to software or hardware already available from a commercial vendor.
CSC	Computer Sciences Corporation
CSOC	Consolidated Space Operations Contract. A contract with NASA to provide space operations support. Lockheed-Martin, Honeywell, and Computer Sciences Corporation (CSC) are the three largest contractors involved.
DDM	Data Distribution Management. The HLA-defined RTI service, which controls the distribution and visibility of shared data within a federation.
discover	Whenever a federate registers a new instance of a class with the RTI, the RTI informs the other federates which have subscribed to the class. When a federate receives notification of a new object instance of interest, it is said in HLA terminology to “discover” it.
DMSO	Defense Modeling and Simulation Office. An organization within the DoD responsible for simulations and modeling.
DoD	[U.S.] Department of Defense. The department of the U.S. Government responsible for the nation’s defense and the management of its armed forces.
federate	An individual simulator application making up one component of a larger and often distributed simulation system (federation).
Federate ambassador	A base class provided with the RTI software. Consists of a collection of virtual member functions to be implemented by the specific federate applications. The RTI evokes these functions as callback routines in order to communicate with the federate application.

federation	A collection of HLA-compliant simulator applications (federates) which are collaboratively integrated together into a larger simulation system.
FOM	Federation Object Model. The definition of the types and forms of all data shared by federates within a federation. The definitions of all of the object and interaction classes known to the federation makeup the FOM. The FOM must be documented in accordance with the HLA specification.
FOM object	An HLA object class defined in the FOM. Note that this is not the same as a C++ object.
"FOM-O-Rama"	The integration meeting between the persons responsible for each federate where they reconcile the SOMs of their respective federates together into a federation-wide FOM. The meeting where the FOM is agreed upon and defined.
FreeFlyer®	A commercial product sold by AI Solutions, Inc., which provides space mission modeling and flight dynamics computations.
FSF	Free Software Foundation. A non-profit organization based in Boston, MA, which develops free and open-source software. The FSF raises money for and manages the GNU project.
GNU	"GNU is Not Unix." A recursive acronym, which refers to the free and open-source software project managed by the non-profit Free Software Foundation. The GNU project developed major portions of the Linux operating system.
GOTS	Government Off-The-Shelf. Software that was developed on a U.S. Government contract, is owned by the U.S. Government, and is available for use without cost to other Government projects.
GPL	GNU General Public License. A software license defined by the FSF that that permits the free use, copying, and redistribution of software and source code.
GSFC	Goddard Space Flight Center. A NASA facility in Greenbelt, Maryland.
HLA	High Level Architecture. A standard architecture for simulations and modeling developed by DMSO also known as IEEE-1516.
IEEE	Institute of Electrical and Electronics Engineers. A professional organization, which defines and publishes standards in the computing and engineering fields.
IEEE-1516	An industry standard for simulations and modeling architectures based on the HLA.
interaction	An HLA component which contains ephemeral data communicated between two or more federates; essentially a broadcast message.
Linux™	A free and open source Unix-like operating system licensed under the GPL.

LRC	Local RTI component. The component of the RTI which is bound to a federate and allows the federate to communicate with the RTI ambassador.
Matlab®	A commercial software product soled by The MathWorks, Inc.™ Essentially, a fourth generation language (4GL) and programming environment for mathematics, scientific and other technical applications, and data display.
NASA	National Aeronautics and Space Administration. The Agency of the U.S. Government primarily responsible for space exploration and space operations.
object	An HLA structure, which contains persistent data shared between two or more federates within a federation, a FOM object. Also, a type of data container provided by the OOP paradigm that associates data with the functions that operate on them.
Octave	A free, fourth generation language (4GL) and programming environment for mathematics and technical applications that is mostly source compatible with Matlab® and is licensed under the GPL.
OMT	Object Model Template. A standard template for defining and documenting FOMs and SOMs.
OOP	Object-Oriented Programming. A software design paradigm that bundles data and the functions that operate on them together into single, discreet components from which other more specialized components may be derived.
parameter	An individual data item contained within an HLA FOM interaction.
pure virtual function	A member function that is declared in the specification of an abstract base class but not implemented in that base class. The implementation is deferred to the derived classes. All derived classes must implement the function.
POSIX	Portable Operating System Interface. An industry standard for interfacing to Unix and Unix-like operating systems.
publish	The act of a federate informing the RTI that it intends to update the attributes of a particular class of FOM object.
register	The act of a federate creating a new instance of a FOM object class through the RTI.
reflect	The act of a federate reading data values from an FOM object instance's attributes after another federate has updated them.
RTI	Runtime Infrastructure. The software which implements the HLA interface and manages the simulation.
RTI ambassador	The class contained in a library provided with the RTI software, which contains public member functions called by the federate to communicate with the RTI.

SAIC	Science Applications International Corporation. SAIC is the prime contractor to DMSO for the development and support of the DMSO RTI.
SIMSS	Scalable Integrated Multi-mission Simulation Suite. A customizable and configurable spacecraft simulation and telemetry generation system.
SOM	Simulation Object Model. The definition of the types and forms of all data shared by a given federate with the federation. The SOM must be documented in accordance with HLA specifications.
STK®	Satellite Tool Kit. A commercial product sold by Analytical Graphics, Inc.™ (AGI), which provides extensive space mission modeling and visualization features.
subscribe	The act of a federate informing the RTI that it intends to read the attributes of a particular class of FOM object and that it wishes to be informed whenever those attributes are updated.
TLE	Two-Line Elements. A set of parameters, which describe the orbit of a satellite, from which the position and velocity of the satellite can be computed for any given time.
UML	Unified Modeling Language. A standard notation for documenting software and system designs, work flows, and industrial processes.
virtual function	A member function in a base class, which can be overridden by an identically declared member function in a derived class.